Optic Algebras: Beyond Immutable Data Structures

J. Lopez-Gonzalez[a,b,1,*], J.M. Serrano[a,b]

[a] *Universidad Rey Juan Carlos*
[b] *Habla Computing*

Abstract

Abstract goes here!

1. Introduction

Optics provide abstractions and patterns to access and update immutable data structures. Specifically, they access and update *focus* values which are contextualized within a *whole* data structure. There are different kinds of optics in the folklore[2], each of them describing a particular type of transformation. Among them, *lens* has become the most prominent one.

A lens do focus on a unique value that is always available from the whole data structure. They turn out to be very useful to express transformations over nested data structures [1]. As an example, consider the following Scala classes [2], where we use macro annotations[3] to generate a lens for each field (or focus) of the annotated case class (or whole):

```
package data

@Lenses case class Person(
  name: String,
  address: Address)

@Lenses case class Address(
  city: String,
  zip: Int)
```

Now suppose we want to modify the zip code associated to a person. Since lens composition is closed [3], we could express that transformation by combining the lenses associated to *address* and *zip* fields and then using the resulting one to invoke the modifying method:

```
def modZip(f: Int ⇒ Int): Person ⇒ Person =
  (address compose zip).modify(f)
```

Despite the elegance that we get to describe this kind of transformations, lenses are restricted to work exclusively with in-memory data structures. Imagine that we wanted to persist this person in a relational database using *Doobie* or *Slick*. In this scenario, we would discard optics in favor of the specific transformations provided by these technologies. Otherwise, we would need to pull the whole state, modify it by means of a lens and finally put it back again, which turns out to be completely impractical.

*Corresponding author
Email addresses: j.lopezgo@alumnos.urjc.es (J. Lopez-Gonzalez), j.serrano.hidalgo@urjc.es (J.M. Serrano)

[1] This is the specimen author footnote.
[2] http://oleg.fi/gists/images/optics-hierarchy.svg
[3] This annotation is provided by *Monocle*, the most relevant optic library in the Scala ecosystem.

Firstly, we claim that it is possible to collect the essence of optics in a domain specific language [4], aimed to take optic abstractions and patterns beyond in-memory data structures. Secondly, we claim that this language is expressive enough to implement the data layer of an application and its associated logic once and for all, and later instantiate them to particular state-based interpretations: in-memory data structures, databases, microservices, etc. Lastly, we claim that we can do so while retaining reasonable levels of performance for a large class of applications. These are our contributions to fulfill our claims:

- We propose *lens algebra* (section 2), which distills the algebraic essence of a lens. To better illustrate this idea, we need to see lenses from a coalgebraic perspective [5, 6, 7]. In this sense, optics are just machines —that can possibly run forever [8]— with buttons that can be pressed to observe or forward the machine state. In fact, what we mean by the algebraic essence is just the description of the machine interface in terms of its buttons, where any reference to the inner state has been abstracted away.
- Lens algebras in their *raw* formulation do not compose. In fact, it is difficult to determine what we mean by composition at this level of abstraction. To alleviate the situation, we propose a *natural* representation (section 3) —which heavily relies on natural transformations [9]— that enables composition for lens algebras.
- We take these ideas to other optics, particularly: *getter*, *setter*, *fold*, *affine* and *weak traversal*[4]. As a result, we end up with a collection of algebras that we refer to as *optic algebras* (section 4). These algebras represent analogous transformations to their in-memory optic counterparts, even preserving heterogeneous composition, but in a more general setting.
- We provide *Stateless* (section 5), an optic algebra library which is aimed to take these concepts and patterns to the development of industrial applications in the Scala ecosystem.

Previously, when we stated that optic algebras pursue transforming state in different settings, we actually meant that we want them to support different state-based effects [10]. In this sense, there exists work relating optics and effects, being *monadic lenses* [11] the most relevant one. Bidirectional transformations [12, 13] is also a great source of effectful transformations. We will compare our approach with these techniques in section 6.

2. Towards more abstract lenses

As stated in [1], lenses approach the view update problem for tree-structured data.

Definition 2.1. *A* lens *consists of a pair of functions, one of them* gets *the part from the whole and the other one* sets *a new part into an old whole, returning a new whole. They are encoded in Scala as follows [14]:*

```
case class Lens[S, A](
  get: S => A,
  set: (S, A) => S)
```

As you may figured out, S and A serve as the whole and the part, respectively. Lens operations must obey certain laws to be considered very well-behaved [15]:

$$set(s,\ get(s)) = s \qquad \textbf{GetPut}$$
$$get(set(s,\ a)) = a \qquad \textbf{PutGet}$$
$$set(set(a1,\ s),\ a2) = set(s,\ a2) \qquad \textbf{PutPut}$$

In section 1 we saw that lens generation for the fields of a case class is systematic, since we generated them automatically. Anyway, here it is the manual implementation of *zip*, which just determines what is the part that we may want to access or update:

[4]This is a relaxed traversal, where the transformation function for each focus must be exactly the same.

```
val zip = Lens[Address, Int](
  s => s.zip,
  (s, a) => s.copy(zip = a))
```

Now, it is time for us to distill the algebraic essence of a lens, since we are interested on taking its interface to new settings. To do so, we need to view this abstraction from a coalgebraic perspective [5, 6]. Particularly, coalgebras can be seen as machines with buttons to forward its current state. These buttons are special, since they could require an input to be pressed or they could produce an output to the environment. The following process turns the original lens operations into state-based transformations were buttons are made explicit.

Lemma 2.1. *Let $s_0, s_1 \in S$ and $a_0, a_1 \in A$. If we consider a very well behaved lens (*get, set*) with whole S and focus A, where s_0 and s_1 enclose a_0 and a_1, respectively*

$$
\begin{aligned}
& (get, set) \\
&= (s_0 \Rightarrow a_0,\ (s_0,\ a_1) \Rightarrow s_1) && \textit{(expand methods)} \\
&= (s_0 \Rightarrow a_0,\ (a_1,\ s_0) \Rightarrow s_1) && \textit{(flip set arguments)} \\
&= (s_0 \Rightarrow a_0,\ a_1 \Rightarrow s_0 \Rightarrow s_1) && \textit{(curry set)} \\
&= (() \Rightarrow s_0 \Rightarrow a_0,\ a_1 \Rightarrow s_0 \Rightarrow s_1) && \textit{(input for get)} \\
&= (() \Rightarrow s_0 \Rightarrow a_0,\ a_1 \Rightarrow s_0 \Rightarrow (s_1,\ ())) && \textit{(output for set)} \\
&= (() \Rightarrow s_0 \Rightarrow (s_0,\ a_0),\ a_1 \Rightarrow s_0 \Rightarrow (s_1,\ ())) && \textit{(resulting state for get)} \\
&= (() \Rightarrow \mathit{State}(s_0 \Rightarrow (s_0,\ a_0)),\ a_1 \Rightarrow \mathit{State}(s_0 \Rightarrow (s_1,\ ()))) && \textit{(State wrap)}
\end{aligned}
$$

So, we started with the pair of lens operations and we ended up with a pair of buttons, each of them showing the notion of input, output and state transformation. Particularly, to homogenize the button intuition in both operations, we had to provide an artificial input to *get*, an artificial output to *set* and an artificial resulting state to *get* —which keeps the old state as is. Thereby, we could represent the type that describes the final buttons in this interface as follows:

```
trait LensButtons[S, A] {
  def get(): State[S, A]
  def set(a: A): State[S, Unit]
}
```

Despite the fact that we identified the buttons, it is still difficult to see how to take them to other settings. Indeed, LensButtons is completely coupled to in-memory data structures. This is evidenced in the right hand side of the methods, where we can find plain State programs. If we aim at being more generic, we should abstract away any reference to State[S, ?] from the buttons.

Definition 2.2. *A lens algebra is a multi-parameter typeclass that is indexed over two type parameters F[_] and A and which defines a pair of methods get and set.*

```
trait LensAlg[F[_], A] {
  def get(): F[A]
  def set(a: A): F[Unit]
}
```

Informally, we can see that F is representing a state-based effect, which let us access or update a value typed A —which is somehow contextualized in F— by means of get and set.

Remark. *We can recover LensButtons by instancing LensAlg for State[S, ?]*

```
type LensButtons[S, A] = LensAlg[State[S, ?], A]
```

Most of typeclasses require laws to be useful, and LensAlg is not an exception for that. Since the original lens holds several properties, we could infer that LensAlg laws will be related to them somehow. Take lens *GetPut* law as an example. It establishes that getting the part and then setting it again does not produce an overall effect. It is now tempting to think that LensAlg should behave accordingly. However, get returns a value typed F[A], which cannot be fed to set.

Monad is the combinator that we need to recover lens laws in LensAlg. If F is an instance of monad, we can establish the analogous *GetPut* for LensAlg as follows:

$$get >>= set = return () \qquad \textbf{GetPut}$$

We can represent the other laws using monadic analogous as well. In addition, we need to include a new law, which emerges when we represent get as a stateful computation. This new law tells us that this method cannot modify the machine state, since it is a mere observer. This typeclass, laws included, should be familiar to a functional programmer.

Corollary 2.1.1. *If you distill the algebraic essence of a lens, you get MonadState (laws included).*

We, as programmers, are used to work with the instance MonadState[State[S, ?], S], where the value behind State is precisely the value that you have access to from the typeclass methods. However, we gave a different intuition for lens algebras, where we exhibited the instance LensAlg[State[S, ?], A], where A is somehow contextualized in S. Surprinsingly, in order to implement such an instance, we are going to need a Lens[S, A]. In fact, there are many interesting connections between lenses and state computations [12].

Now, we could use lens algebras to represent *address*, from the example of the introduction. Concretely, we could express it as a pair of lens algebras, one of them granting access to the city and the other one doing the proper for the zip code, as follows:

```
trait Address[Q[_]: Applicative] {
  def city: LensAlg[Q, String]
  def zip:  LensAlg[Q, Int]

  def serialize: Q[String] = (city.get |@| zip.get) { (c, z) =>
    s'{ "city": $c, "zip": $z }'
  }
}
```

As you can see, Address is itself a typeclass. It is defined for every F, as long as we can provide the pair of lens algebras city and zip for it. The intuition is that Q is somehow hiding the means to manipulate the state of the address.

The major benefit that we get by adopting this pattern is that we can implement the business logic once and for all, exploiting the same algebra that we use when dealing with normal lenses. This is evidenced in serialize , which returns a program that invokes get to collect and serialize the involved values.

We saw that optic composition is a key feature to cope with nested data structures. In the next section, we will cover composability at this high level of abstraction enabled by lens algebras.

3. Natural lens algebras

A particular lens is not very useful by itself, but it becomes extremely handy in combination with other lenses. Since concrete lenses do not compose well, there has been great interest on discovering new lens representations —such as *van laarhoven* or *profunctor*— to improve this feature [3]. In fact, these representations compose nicely, because they are essentially functions.

There is another function-like representation for lenses which is not so widespread in the folklore [12]. It consists of a particular monad morphism[5]:

[5]We use *kind-projector* to dulcify the morphism types

```
type Lens[S, A] = State[A, ?] ~> State[S, ?]
```

Informally, if you have a state program on the focus, you can transform it into a state program on the whole. As any other homomorphism, this natural transformation is composable. Thereby, we could compose it with another lens, as long as its whole matches the focus of the original lens. What is interesting about this particular representation is that a state program produces an output value along with the state transformation. This idea will serve us well later.

As expected, we are interested on taking composition to optic algebras. Particularly, we will exploit the last lens representation to do so, but prior to that, we need to understand what we are referring to when we talk about lens algebra composition. Using the guiding example, we could define a person lens algebra as we did for address in the last section, aiming at composing their inner lenses:

```
trait Person[P[_]] {
  type Ad
  def address: LensAlg[P, Ad]
}

trait Address[Q[_]] {
  def zip: LensAlg[Q, Int]
}
```

So, we get a Person which is parameterized by P, the type of programs that knows how to evolve a person. This trait contains a lens address, whose focus is Ad, we are not sure about the particular representation for it, so we left it opened. Now, we are interested on composing it with zip, as we did previously with concrete lenses. To recap, there is zip lens, which produces programs Q to evolve an address, and there is address lens, which produces programs P to evolve a person. If we want to compose both lenses, we need to inject Q programs into P ones. This sounds a lot like a natural transformation Q ~> P.

As a naive approach, we could consider that a monad morphism from Q to P is a nice candidate to be a composable representation for optic algebras. In fact, Q ~> P is what we get when we abstract away from State[A, ?] ~> State[S, ?]:

```
trait NaiveLensNat[P[_]: Monad, Q[_]: Monad, A] {
  def apply: Q ~> P
}
```

However, it is not possible to implement get and set given this monad morphism alone, so it is difficult to establish connections with the results obtained from section 2.

To avoid this situation, we need to relate somehow the type constructors P and Q with the focus A. Indeed, when we abstract State[A, ?] from the monad morphism, we can determine not only that Q is a monad but also that it is a MonadState for A. It appears that if we add this new constraint to our monad morphism, we are able to implement get and set, feeding the corresponding programs to the homomorphism:

Definition 3.1. *The natural representation of a lens algebra is a monad morphism which is lifted into a monad state morphism from Q —the inner program that evolves the part— to P —the outer program that evolves the whole.*

```
trait LensNat[P[_]: Monad, Q[_]: MonadState[A, ?[_]], A]
    extends LensAlg[P, A] { self =>

  def apply: Q ~> P

  def get: P[A] = apply(MS.get)
  def set(a: A): P[Unit] = apply(MS.put(a))
}
```

By implementing get and set this way, we are actually forcing the monad morphism to be a *monad state morphism*. In other words, we are turning Q —a program that evolves the focus A— into P —a program

that evolves the whole (hidden by P)— but the morphism preserves get and set, so we grant access to A in terms of P programs.

The gretatest benefit we get by using LensNat is that it enables lens algebra composition. Next, we show the corresponding method to achieve such task:

```
def compose[R[_], B](other: LensNat[Q, R, B]): LensNat[P, R, B] =
  new LensNat[P, R, B] {
    def apply: R ~> P = self.apply compose other.apply
  }
```

This tells us that if we compose a lens ln1: LensNat[P, Q, A] with another lens ln2: LensNat[Q, R, B] we get a new lens algebra typed LensNat[P, R, B]. To put it in another way, if the inner program of ln1 matches the outer program of ln2 we can define a new lens algebra whose inner program focuses in a more specific part of the state.

Now we should be able to compose address with zip. To do so, we need to establish a connection between the inner program from address and the outer program from zip. We could do so using the next pattern:

```
trait Person[P[_]] {
  type Ad
  type Q[_]: Address
  def address: LensNat[P, Q, Ad]
}

trait Address[Q[_]] {
  def zip: LensNat[Q, State[Int, ?], Int]
}
```

Notice that we had to specify that Q instantiates the Address typeclass to connect address inner program with zip outer one. On the other hand, the inner program for zip is concrete, since we are not interested on further composing this particular lens algebra.

Once we have defined the basic data layer in general terms, we can use it to implement the application logic. In this particular scenario, we could be interested on modifying the zip code associated to a person:

```
def modZip[P[_]](f: Int ⇒ Int)(p: Person[P]): P[Unit] = {
  import p.address, p.Address.zip
  (address compose zip).modify(f)
}
```

As you can see, the implementation is completely decoupled from any infrastructure. It no longer works for a specific case class, but for any type P that qualifies a Person. And, still, what is great here is that this implementation is almost the same as the one we did for the in-memory scenario in section 1.

4. Generalizing to other optics

Lens composition is fundamental to deal with nested data structures, but optics really shine when they are composed heterogeneously [3]. Therefore, in order to take optic benefits to the algebraic setting, we need to describe other optic algebras and be able to compose them. In this section, we introduce *affine algebras* and *traversal algebras*. For each of them, we will show a very brief introduction, we will extract its algebraic essence and finally, we will show a composable representation.

4.1. Affine Algebra

A lens implies that the part is always present in the whole, however this could not be the case in other scenarios. Affine[6] comes to alleviate this situation by evidencing that the part could be unavailable from the whole:

[6]This optic is also known as *affine traversal*, *affine lens* or even *optional* in the folklore.

```
case class Affine[S, A](
  getOption: S => Option[A],
  set: (S, A) => S)
```

If we derive the essence of the affine, as we did in section 2 for lens, we end up with the following definition for affine algebras:

```
trait AffineAlg[F[_], A] extends Monad[F] {
  def getOption(): F[Option[A]]
  def set(a: A): F[Unit]
}
```

As for the lens algebra case, it does not make much sense to compose affine algebras. We need to find a morphism representation to enable this feature. Intuitively, an inner program that modifies the part should be transformed into an outer program that optionally modifies the whole, just in case the part does exist. This intuition lead us to the following representation:

```
case class NatAffineAlg[P[_], Q[_], A](
    hom: Q ~> P[Option[?]])(implicit
    MS: MonadState[A, Q]) extends AffineAlg[P, A] {
  def getOption(): P[Option[A]] = hom(MS.get)
  def put(a: A): P[Option[Unit]] = hom(MS.put(a))
}
```

In fact, if we instantiate this algebra for NatAffineAlg[State[S, ?], State[A, ?], A] we get something isomorphic to Affine[S, A], which supports the new algebra definition. Now that we have a composable representation, we should be able to compose a pair of affine algebras:

```
def compose[P[_], Q[_], R[_], B](
    af1: NatAffineAlg[P, Q, A]
    af2: NatAffineAlg[Q, R, B])(implicit
    MS1: MonadState[A, Q],
    MS2: MonadState[B, R]): NatAffineAlg[P, R, B] =
  NatAffineAlg((af1.hom compose af2.hom).map(_.join))
```

And what is even more interesting, we should be able to compose optics heterogeneously, for instance, a lens algebra with an affine algebra:

```
def compose[P[_], Q[_], R[_], B](
    ln: NatLensAlg[P, Q, A]
    af: NatAffineAlg[Q, R, B])(implicit
    MS1: MonadState[A, Q],
    MS2: MonadState[B, R]): NatAffineAlg[P, R, B] =
  NatLensAlg(ln.hom compose af.hom)
```

This composition results in an affine algebra, which is pretty consistent with classic optic composition, where the result of composing a lens with an affine is itself an affine. Notice as well that reversing the composing optic types, eg: composing an affine algebra with a lens algebra, does not affect the resulting type.

4.2. (Weak) Traversal Algebra

While lenses and affines do focus on a single part, which could or could not exist, we lack something to focus on a sequence of parts, which is exactly what traversals are for. This optic turns out to be tricky and it is not possible to represent it as a bunch of simple functions [7]. Therefore, we will use a relaxed version, which we call *Weak Traversal*:

```
case class WTraversal[S, A](
  getAll: S => List[A],
  modify: (A => A) => S => S)
```

This optic is constrained in the sense that the modification that we achieve over each part must be exactly the same, but we find it good enough for most of cases. If we derive the essence of a weak traversal, we get the following interface:

```
trait WTraversalAlg[F[_], A] extends Monad[F] {
  def getList(): F[List[A]]
  def modify(f: A ⇒ A): F[List[Unit]]
}
```

As usual, we need to find a composable representation for these traversal algebras. If we apply the same ideas that we used for affine algebras, we get the next representation, where the possibility of failing focus is replaced with the possibility of multiple focus:

```
case class NatWTraversalAlg[P[_], Q[_], A](
    hom: Q ~> P[List[?]])(implicit
    MS: MonadState[A, Q]) extends WTraversalAlg[P, A] {
  def getList() = hom(MS.get)
  def modify(f: A ⇒ A) = hom(MS.put(a))
}
```

Again, if we instantiate this algebra for NatWTraversalAlg[State[S, ?], State[A, ?], A] we get something isomorphic to WTraversal[S, A]. Now that we have a composable representation, we should be able to compose homogeneously:

```
def compose[P[_], Q[_], R[_], B](
    af1: NatWTraversalAlg[P, Q, A]
    af2: NatWTraversalAlg[Q, R, B])(implicit
    MS1: MonadState[A, Q],
    MS2: MonadState[B, R]): NatWTraversalAlg[P, R, B] =
  NatWTraversalAlg((af1.hom compose af2.hom).map(_.join))
```

And besides, we should be able to compose optics heterogeneously, for instance, a lens algebra with a traversal algebra:

```
def compose[P[_], Q[_], R[_], B](
    ln: NatLensAlg[P, Q, A]
    af: NatTraversalAlg[Q, R, B])(implicit
    MS1: MonadState[A, Q],
    MS2: MonadState[B, R]): NatTraversalAlg[P, R, B] =
  NatTraversalAlg(ln.hom compose af.hom)
```

As expected by optic composition rules, composing lens algebras with traversal algebras results in traversal algebras. Had we composed with an affine algebra instead, it would have resulted in a traversal algebra.

5. Stateless: an optic algebra Scala library

Once we have introduced the fundamentals of optic algebras, it is time to put them into practice. In fact, we have implemented Stateless, a Scala library where we deploy algebras and utilities to deal with them. We will use a guiding example to introduce the library basics. Particularly, we will extend the *zip code* one. Now, we will consider that there exists a department, with a budget and a list of members. For each member, we will be able to access its name and optionally, to access its address. Each address, in turn, will contain the associated city and zip code. Our objective is to implement modifyZip, that receives a function to modify a zip code and returns a program that modifies all the zip codes belonging to the people in the department according to the input function.

Our example contains three major entities: address, person and department. Each of them will be located in its own data structure. We will start with address, which contains the name of the city and the zip code itself. Since they appear exactly once in every address, we know that accessing those fields could be achieved by means of lenses. Therefore we represent addresses as follow:

```
trait Address[Ad] {
  type P[_]
  val city: LensAlg[P, String]
  val zip: LensAlg[P, Int]
}
```

First of all, this trait takes a type parameter that represents the final state of the address, or the context to get access to it. Secondly, it contains a **type** member which roles programs that are able to access and modify the address state. Finally, you can find a lens algebras for each field. Concretely, this is a natural lens algebra. Contrary to what we saw in section 3, Stateless use the same name for both raw and natural optic algebras (which are kept in different packages), and hides the Q type parameter as a type member for the second case, to improve compiler inference while composing optics. Now, we show how to encode a person:

```
trait Person[Pr] {
  type P[_]
  type Ad
  val name: LensAlg[P, String]
  val address: Address[Ad]
  val optAddress: OptionalAlg.Aux[P, address.P, Ad]
}
```

As before, this entity takes a type parameter Dp, which represents the final state, and a type member P, which corresponds with the program that transforms the state. Besides, this entity contains two fields and therefore they are mapped into two optic algebras. There is nothing remarkable about name, but optAddress brings new patterns. Specifically, its returning type contemplates Q, since we need to connect its inner program with the outer program from Address, to make them composable. As a consequence, we need to provide an evidence of entity address, and therefore the type of its final state Ad. The last entity, department, is represented as folows:

```
trait Department[Dp] {
  type P[_]
  type Pr
  val budget: LensAlg[P, Long]
  val person: Person[Pr]
  val people: TraversalAlg.Aux[P, person.P, Pr]
}
```

It contains two optic algebras, a simple lens for budget and a dependent traversal for people. Concretely, it has to establish the dependency between department and person. To do so, we use the same pattern we described above for optAddress.

The resulting data layer does not differ greatly from representing the state of the application with several case classes. In this sense, a simple field is represented with a lens algebra, an optional field is represented with an affine algebra and a list of elements is represented with a traversal algebra.

The most striking feature of this data layer is that we can describe modifyZip once and for all, in a modular and elegant way:

```
def modifyZip(f: Int ⇒ Int): P[Unit] =
  (people compose optAddress compose zip).modify(f)
```

As you see, Stateless contains infix methods to compose a particular optic with another passed as argument, following the classic optic hierarchy. In this regard, it also contains abstracting methods, to turn an optic algebra into a hierarchy ancestor. Finally, it also provides the means to transform an optic algebra into its corresponding indexed one.

Notice that modifyZip knows nothing about the particular programs or the final application state. In fact, they could be mere *State* programs dealing with case classes as state, or they could be *IO* programs using a database session as state. We show these particular interpretations in the appendix. There you will find the utilities that Stateless provides to facilitate the instantiating task.

6. Related Work

7. Conclusion

Acknowledgements

MINECO

References

[1] J Nathan Foster, Michael B Greenwald, Jonathan T Moore, Benjamin C Pierce, and Alan Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. *ACM SIGPLAN Notices*, 40(1):233–246, 2005.

[2] Martin Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[3] Matthew Pickering, Jeremy Gibbons, and Nicolas Wu. Profunctor optics: Modular data accessors. *arXiv preprint arXiv:1703.10857*, 2017.

[4] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, 28(4es):196, 1996.

[5] Bart P.F. Jacobs. Objects and classes, coalgebraically. Technical report, Amsterdam, The Netherlands, The Netherlands, 1995.

[6] Bart Jacobs and Jan Rutten. A tutorial on (co) algebras and (co) induction. *Bulletin-European Association for Theoretical Computer Science*, 62:222–259, 1997.

[7] Russell O'Connor. Functor is to lens as applicative is to biplate: Introducing multiplate. *arXiv preprint arXiv:1103.2841*, 2011.

[8] David A Turner. Total functional programming. *J. UCS*, 10(7):751–768, 2004.

[9] Benjamin C Pierce. *Basic category theory for computer scientists*. MIT press, 1991.

[10] Philip Wadler. Monads for functional programming. In *International School on Advanced Functional Programming*, pages 24–52. Springer, 1995.

[11] Faris Abou-Saleh, James Cheney, Jeremy Gibbons, James McKinna, and Perdita Stevens. Reflections on monadic lenses. *CoRR*, abs/1601.02484, 2016.

[12] Faris Abou-Saleh, James Cheney, Jeremy Gibbons, James McKinna, and Perdita Stevens. Notions of bidirectional computation and entangled state monads. In *International Conference on Mathematics of Program Construction*, pages 187–214. Springer, 2015.

[13] Faris Abou-Saleh and Jeremy Gibbons. Coalgebraic aspects of bidirectional computation. *Journal of Object Technology*, 16(1), 2017.

[14] Tony Morris. Asymmetric lenses in scala. 2012.

[15] Sebastian Fischer, Zhenjiang Hu, and Hugo Pacheco. A clear picture of lens laws. In *International Conference on Mathematics of Program Construction*, pages 215–223. Springer, 2015.